B K for the Maude User

In this section we discuss two approaches that one can use to incorporate K language definitions in Maude. The first approach makes plain use of Maude and should be easy to translate and use in the context of any other rewrite engine. Also, the first approach is slightly easier to understand than the second one. The second approach makes use of Maude's special reflective capabilities using its provided meta-level; at our knowledge Maude is the only rewrite engine providing such capabilities. In spite of the apparently heavier syntax notation that it involves and its slightly lower performance (when executed) in Maude, the second approach has a series of advantages over the first one, explained in Section **??**, which currently make it our prefered approach. Nevertheless, if the language that one wants to define does not include (rather complex) features that involve code generation or reflection, then the first approach is just as good. In fact, the only thing one looses when one uses the first approach for such complex language features is modularity: one needs to change its definition when moving the feature from one language to another, because the syntax to be used in the generated code of the language changes. This non-trivial sort of modularity can be achieved very elegantly in K, thanks to its uniform representation of computations and because of its deliberate decision to disobey the syntax. Both approaches are relatively straightforward and mechanical.

We discuss the two approaches separately, using for concreteness' sake the simple imperative language in Figure 2. We have developed, for each of the approaches, a little "library", or "prelude", of K-related definitions that turned out to be useful in our experiments. These libraries include definitions for lists, sets, mappings, locations, environments and stores, etc. Before starting a new language definition, one should include the specific prelude file. Both our current prelude files, which may change in the future, are discussed in Section **??**; however, one needs not understand all the details of how they are defined in order to use them. All one needs to know is what one can use from them. For example, they both provide a sort `K`, together with subsorts `KResult` and `KProper`; the former corresponds to computations that are finished and therefore contain no further computational tasks (such as values, or types, or results of analysis, etc.), while the latter corresponds to proper (and well-formed!) computations that still contain computational tasks to be processed. In both approaches, proper computations can be further "heated", while results are used for cooling and for matching purposes in order to apply the actual, irreversible rewrite rules.

B.1 Using Plain Maude

Step 1 Load the provided prelude file for K definitions using plain Maude, called "`k-plain-prelude.maude`":

```
in k-plain-prelude
```

Step 2 Define the syntax of the desired programming language as an algebraic signature, making sure that you include the module `K-PLAIN`. Here is, for example, the syntax of our simple language:

```
fmod IMP-SYNTAX is including INT + BOOL + NAME + K-PLAIN .
  sorts #Int #Bool #Name .
  op #int : Int -> #Int .
  op #bool : Bool -> #Bool .
  op #name : Name -> #Name .
  subsorts #Int #Bool < K .
  subsort #Name < KProper .
  op _+_ : K K -> KProper [prec 33] .
  op _<=_ : K K -> KProper [prec 37] .
  op _and_ : K K -> KProper [prec 55] .
  op not_ : K -> KProper [prec 53] .
  op skip : -> KProper .
  op _:=_ : K K -> KProper .
  op _;_ : K K -> KProper [prec 100 gather(e E)] .
  op if___ : K K K -> KProper .
  op while__ : K K -> KProper .
  op halt_ : K -> KProper .
--- kept only one ``_;_'', because the two have the same computational meaning
endfm
```

One should be aware of several requirements and guidelines here:

- Most importantly, *one should use only one language syntactic category, K*. The sort `K` is already provided in "`k-plain-prelude.maude`", so one does not need to redeclare it. In other words, for our language in Figure 2, we collapse `AExp`, `BExp`, `Stmt` and `Pgm` into only one sort `K`, which will be the only one visible from here on. As already mentioned, the sort `K` comes with an important subsort, `KProper`, which, intuitively, stays for those computations that need to be further processed to become results. It is important to declare all the result sorts of language constructs which are not already values as `KProper`. No computation constructs need to be declared as constructs for `KResult` when defining the language syntax. The sort collapsing process described above may certainly appear to be very inconvenient because one can now write programs which are not well formed, such as "$3 + \texttt{true}$". Nevertheless, at this moment one should think of the syntax of the language as a syntax for its *abstract syntax tree (AST)*. In other words, its syntax at this stage is nothing but a list of AST node labels. When defining real languages for research or prototyping purposes, one may want to implement an external parser, using the state of the art parsing technology, which takes as input a program following the desired user-friendly syntax and generates as output an AST following the simplified syntax that we define in Maude. As seen in Section **??**, one can also define type-checkers using the same K technique, in which case one can reject programs which are regarded as innapropirate (for typing or other safety reasons).
- A good practice is to *wrap the builtin sorts imported in the syntax of the language*, as well as those that one may use as an intrinsic part of the semantics later on. For example, we import both integers and booleans in our language. Since we collapsed the different language syntactic categories into just `K`, then, without an appropriate separation, the various imported sorts would also collapse, which may lead to many Maude warnings and errors. For example, one cannot even put together integers and booleans under a common sort in Maude, because it just happens that the two modules have operators sharing the same name and number of arguments. Maude rightfully complains, because one cannot always know from context which operator one refers to. Our convention is to use a wrapper name "`#sort`" for each sort that one wants "protected" from here on. We protect three such sorts in our language syntax: `Int`, `Bool`, and `Name`. Names are also protected in almost all our language definitions, because we are going to manipulate them as part of the semantics (save in environments, states, etc.). For clarity, though not stricly needed, one may define subsorts of `K` corresponding to the imported sorts, such as `#Int`, `#Bool`, etc.
- Since there is only one syntactic category, *one may have to either remove or redeclare some original language constructs*. For example, in our original syntax we had two semicolon constructs, *Stmt*; *Stmt* and *Stmt*; *AExp*, one for statements and the other for programs. Fortunately, these two constructs have the same semantics as computations: process the first computation and then the second. Therefore, we only need to add one operation "`_;_ : K K -> KProper`". If the two semicolon constructs had different semantics as computations, then one would have had to define them with different names as computation constructs.

Note that we also added precedence and gathering attributes to some of the language/computation constructs. Those attributes are used by Maude's internal parser and are given exclusively for easying the reading of programs by not having to write some obvious parenthese. The lower the precendence the tighter the binding of that operator. Gathering attributes can specify, among other things, left or right associativity of operators; fore example, we defined the sequential composition to be right associative (left associativity would have also worked).

Step 3 Test the syntax by asking Maude to parse several programs. For example, the program below calculates the sum of all the numbers between 1 and 1000:

```
parse (
        #name(n) := #int(1000) ;
        #name(s) := #int(0) ;
```

```
        #name(i) := #name(n) ;
        while (not(#name(i) <= #int(0))) (
          #name(s) := #name(s) + #name(i) ;
          #name(i) := #name(i) + #int(-1)
        ) ;
        #name(s)
  ) .
```

This is a good moment to write up many interesting programs that one would like to eventually run in the defined language, because at this moment one shoots two rabbits with one stone doing it: (1) comprehensively tests the syntax, and (2) prepares a benchmark to test the subsequent semantics. It is very convenient to properly "divide-and-conquer" the task of defining a language between defining its syntax and defining its semantics. Once the syntax is done, one can move on to the semantics without having to worry about syntactic details anymore. In our experience with designing languages using K, we found fewer aspects that were more annoying than having to go back and modify the syntax while testing the semantics, just because we were not able to parse certain desirable programs in order to execute them; each change of syntax will almost unavoidably trigger corresponding changes of semantics. It is therefore very important to get the syntax right once and for all at this stage.

Step 4 Define all the structural computation equations corresponding to the strictness attributes. These equations can be derived automatically, but we assume no particular translation or implementation of K in Maude here; we simply use Maude to write K language definitions and, obviously, Maude is not aware of K's particularities. The structural computation equations for our language are given below:

```
fmod IMP-K-STRICTNESS is including IMP-SYNTAX .
  vars k k1 k2 : K . vars pk pk1 pk2 : KProper .  vars r r1 r2 : KResult .

  ops (_+[]) ([]+_) : K -> K .
  eq pk1 + k2 = pk1 -> [] + k2 .
  eq r1 -> [] + k2 = r1 + k2 .
  eq k1 + pk2 = pk2 -> k1 + [] .
  eq r2 -> k1 + [] = k1 + r2 .

  ops (_<=[]) ([]<=_) : K -> K .
  eq pk1 <= k2 = pk1 -> [] <= k2 .
  eq r1 -> [] <= k2 = r1 <= k2 .
  eq r1 <= pk2 = pk2 -> r1 <= [] .
  eq r2 -> r1 <= [] = r1 <= r2 .

  op []and_ : K -> K .
  eq pk1 and k2 = pk1 -> [] and k2 .
  eq r1 -> [] and k2 = r1 and k2 .

  op not[] : -> K .
  eq not pk = pk -> not [] .
  eq r -> (not []) = not r .

  op _:=[] : K -> K .
  eq  k1 := pk2 = pk2 -> k1 := [] .
  eq r2 -> k1 := [] = k1 := r2 .

  op if[]__ : K K -> K .
  eq if pk k1 k2 = pk -> if [] k1 k2 .
  eq r -> if [] k1 k2 = if r k1 k2 .

  op halt[] : -> K .
  eq halt pk = pk -> halt [] .
  eq r -> halt [] = halt r .
endfm
```

Note that only proper computations are "heated", while only results are "cooled". As mentioned, the sort *KResult* is a builtin subsort of K, disjoint from *KProper*. The subsequent semantic definitions are going to tell precisely how results are generated and propagated. Also, note that for each strict argument of each language construct we declared a new operation, whose name was that of the original language construct with an apparent "hole" replacing the strict argument. As mentioned, there is no hole in K, but just an intuitive name convention: that's a result "placeholder" for the scheduled subcomputation. The reader is warned that defining the structural computation equations is perhaps the most boring and error prone part of a K language definition. Implementations of K should generate all these automatically from higer-level and compact strictness information.

Step 5 Define the configuration, if needed. With very few exceptions (one example being the trully concurrent semantics of CCS in Section **??**), a K language definition will include some conveniently chosen configuration which will include a set of configuration items. The prelude already defines the sorts `Config` and `ConfigItem`, as well as an operation `[[_]] : KSet{ConfigItem} -> Config`. The sort `KSet{ConfigItem}` is also defined in the prelude; in fact, a parametric module `KSET` is defined for multi-sets (defined using a binary associative and commutative operator), which provides a parametric sort `KSet{X}`, that in this case we instantiate for configuration items. One is free not to use our constructs for configurations if one prefers one's own way to organize them. Here is a configuration definition for our language:

```
fmod IMP-K-CONFIGURATION is including STATE + IMP-SYNTAX + CONFIG .
  op k : K -> ConfigItem .
  op state : State -> ConfigItem .
  subsort Val < Config KResult .
  op [[_]] : K -> Config .
  var p : K .  var v : Val .  var c : KSet{ConfigItem} .
  eq [[p]] = [[k(p) state(.State)]] .
  eq [[k(v) c]] = v .
endfm
```

One should make sure that the language syntax module is included, as well as other modules necessary for the configuration; in our case, we only need to include the modules `STATE` and `CONFIG`, both provided in the prelude. One may need to define new modules if they are not provided in the prelude. `STATE` is defined as an instance of a parametric module `KMAP`, more precisely as `KMAP{Name,Val}`; therefore, values are also automatically included. At this stage in the language definition values are generic, in the sense that we have no constructors for them yet; those will be added when we define the actual semantics of the language. Since values are intended to be results of initial configurations as well as results of computations, this is a good place to state that fact explicitly with the subsort declaration `Val < Config KResult`. To distinguish among the various empty lists or sets for which we generically used a central dot in the K notation, as well as to avoid Maude ambiguous parsing, we suffixed the various "dots" with their corresponding sorts in our prelude (e.g., `.State`, `.K`, etc.) The two equations initiate and terminate the computation process, respectively.

Step 6 Define the actual semantics. The very first step is to "initialize the semantics" by establishing the connection between "terminals" in the syntax, i.e., leaves in the asbtract syntax tree, and computations. Recall that our convention was to wrap those terminals with corresponding operator names, such as `#int`, `#bool`, and `#name`. Our next convention is to use similar wrapper operator names, but dropping the `#` character, for terminals into the computational world; more precisely, we use wrapper operations like "`int : Int -> Val`", "`bool : Bool -> Val`", and "`name : Name -> KProper`". Recall that values are also result computations in `KResult`. The reason for which we declare the first two value wrappers as constructors for values as opposed to constructors for K results is because we want to allow them to be used as values, in particular to be stored in our state structure, for example (in this particular language only integers can be stored, but in general one can store any type of values in a language). The third wraper is a construct for `KProper`, because names still need to be heated

in order to be computed. Expected equations of the form "`#int(i)=int(i)`" establish the desired "bottom-AST" relationship between syntax and computation semantics. This step was not necessary in our "paper" K definitions because there we did not need to wrap AST-terminals at all on paper. The Maude module below shows the complete K semantics of our simple language:

```
mod IMP-K-SEMANTICS is including IMP-K-CONFIGURATION + IMP-K-STRICTNESS .
  vars k1 k2 rest : K . var x : Name . var v : Val . vars i i1 i2 : Int . var b : Bool . var sigma : State .

  op int : Int -> Val .  op bool : Bool -> Val .  op name : Name -> KProper .
  eq #int(i) = int(i) .  eq #bool(b) = bool(b) .  eq #name(x) = name(x) .

  rl k(name(x) -> rest) state(sigma) => k(sigma[x] -> rest) state(sigma) .
  rl int(i1) + int(i2) => int(i1 + i2) .
  rl int(i1) <= int(i2) => bool(i1 <= i2) .
  rl bool(true) and k2 => k2 .
  rl bool(false) and k2 => bool(false) .
  rl not bool(b) => bool(not b) .
  rl k(name(x) := v -> rest) state(sigma) => k(rest) state(sigma[x <- v]) .
  rl k1 ; k2 => k1 -> k2 .
  rl skip => .K .
  rl if bool(true) k1 k2 => k1 .
  rl if bool(false) k1 k2 => k2 .
  eq k(while k1 k2 -> rest) => k(if k1 (k2 -> while k1 k2) .K -> rest) .
  rl k(halt int(i) -> rest) => k(int(i)) .
endm
```

Note that rules use the rule "`rl left => right .`" Maude syntax, denoting the fact that they are, semanticly speaking, irreversible rewrite rules and not equations. The type of the module is `mod` instead of `fmod`, saying that it may contain rewrite rules and thus is not a functional module anymore. Since this particular language happens to be determinisitic, nothing would be lost if all the rules above were equations. However, since we also want to capture the intended computational granularity of each language construct as part of the semantics, in particular that `halt` stops the entire program in *one step*, we prefer to keep them rules.

Step 7 Test the semantics using several programs:

```
rewrite [[
         #name(n) := #int(1000) ;
         #name(s) := #int(0) ;
         #name(i) := #name(n) ;
         while (not(#name(i) <= #int(0))) (
           #name(s) := #name(s) + #name(i) ;
           #name(i) := #name(i) + #int(-1)
         ) ;
         #name(s)
]] .
```

The Maude output for the above reduction, on a 1GH/1GB tablet PC running Windows XP, is:

```
rewrites: 33048 in ... cpu (371ms real) (~ rewrites/second)
result Val: int(500500)
```

Fortunately, when writing the semantic rules above one needs not worry much about forgetting any particular rule. If the benachmarks of collected programs cover all the language constructs, which should always be the case, then one will easily notice the missing rule(s), because the normal-form computation returned by Maude will have the corresponding unreduced language construct at its top. For example, if one omits the first rule, then reducing the term above will give a normal form of the form "`[[k(name(x) -> ...) state(...)]]`", which almost suggests what was forgotten.

B.2 Using Maude's Meta-Level

Step 1 Load the provided K prelude file for definitions using Maude' meta-level capabiliites:

```
in k-meta-prelude.maude
```

Step 2 Define the syntax of the desired programming language as an algebraic signature. Unlike in the previous approach, there is no need to include any K module. K will "swallow" the entire language syntax later at the meta-level. Therefore, we define the language syntax in isolation, as if there was no K:

```
fmod IMP-SYNTAX is including INT + BOOL + NAME .
  sorts #Int #Bool #Name AExp BExp Stmt Pgm .
  op #int : Int -> #Int .
  op #bool : Bool -> #Bool .
  op #name : Name -> #Name .
  subsorts #Name #Int < AExp .
  op _+_ : AExp AExp -> AExp [prec 33] .
  subsort #Bool < BExp .
  op _<=_ : AExp AExp -> BExp [prec 37] .
  op _and_ : BExp BExp -> BExp [prec 55] .
  op not_ : BExp -> BExp [prec 53] .
  op skip : -> Stmt .
  op _:=_ : #Name AExp -> Stmt .
  op _;_ : Stmt Stmt -> Stmt [prec 100 gather(e E)] .
  op if___ : BExp Stmt Stmt -> Stmt .
  op while__ : BExp Stmt -> Stmt .
  op halt_ : AExp -> Stmt .
  op _;_ : Stmt AExp -> Pgm [prec 110] .
endfm
```

As before, one should be aware of several requirements and guidelines here:

- There is no need to collapse the syntactic categories into only one like before, because that will be done automatically at the meta-level when we define the semantics. Even though Maude's parser can reject many ill-formed programs now using its internal parser, one should not expect Maude's parser to parse arbitrarily complex language syntax. Like in the plain Maude approach, if a language designer wants a syntax more complex than what Maude can parse, then she should implement an external parser for the desired syntax, using state of the art parsing technology, to translate it into one that Maude can parse.
- As before, one should also wrap the builtin sorts imported in the syntax of the language, as well as those that one may be used as an intrinsic part of the semantics later on. That will make it easier to recognize these important elements after the program will be lifted at the meta-level. We follow the same wrapper naming conventions like in the plain Maude approach.
- Once the language syntax will be lifted at the meta-level, the operator declarations in the syntax module will be forgotten and their names will be used as node labels in the meta-term (as plain quited identifiers). The sort information of all the language constructs will be discarded; consequently, it is important to rename constructs with the same name but with different sorts in case they have different computational semantics. In our case, we do not need to rename the semicolon constructs *Stmt*; *Stmt* and *Stmt*; *AExp*, because they have the same semantics as computations.

Step 3 Like in the previous approach using plain Maude, test the syntax by parsing several programs. This time, the result sort of well-formed programs will be `Pgm` instead of `KProper`.

Step 4 Define all the structural computation equations corresponding to the strictness attributes. As before, these equations can and should be derived automatically in implementations of K in Maude, but we assume no particular translation or implementation of K in Maude. Some discussion on our meta-term

representation is necessary before we can show the structural computation equations for our simple language. The provided module `K-META` imports the module `K-PLAIN` and "hooks" the Maude internal representation for terms into computations. More precisely, computations can now be constructed with the usual task sequentialization operator `_->_` (corresponding to $\curvearrowright$), with *label constants* of sort `KLabel`, and with terms of the form $L(K_1, K_2, ..., K_n)$, where L is a label and K_1, K_2, ..., K_n are subcomputations. We prefer constants instead of labeled terms with an empty list of children to stay consistent with Maude's meta-term representation conventions. Labels include all the quoted identifiers and can be added more constructs if needed. One builtin label construct, `f : String -> Freezer`, is already provided, the one for "freezers" (`Freezer` is a subsort of `Label`) . A freezer label has therefore the form `f("any sequence of characters")`; the string will most likely include holes, written `[]`. To ease reading of meta-terms with holes, we renamed the Maude's square brackets enclosing meta-subterms to normal parentheses. Here are the structural computation equations for our language:

```
fmod IMP-K-STRICTNESS is including K-META .
  vars k k1 k2 : K .  vars r r1 r2 : KResult . vars pk pk1 pk2 : KProper .

 eq '_+_(pk1,k2) = pk1 -> f(''[']+_)(k2) .
 eq r1 -> f(''[']+_)(k2) = '_+_(r1,k2) .
 eq '_+_(k1,pk2) = pk2 -> f('_+'['])(k1) .
 eq r2 -> f('_+'['])(k1) = '_+_(k1,r2) .

 eq '_<=_(pk1,k2) = pk1 -> f(''[']<=_)(k2) .
 eq r1 -> f(''[']<=_)(k2) = '_<=_(r1,k2) .
 eq '_<=_(r1,pk2) = pk2 -> f('_<='['])(r1) .
 eq r2 -> f('_<='['])(r1) = '_<=_(r1,r2) .

 eq '_and_(pk1,k2) = pk1 -> f(''[']and_)(k2) .
 eq r1 -> f(''[']and_)(k2) = '_and_(r1,k2) .

 eq 'not_(pk) = pk -> f('not'[']) .
 eq r -> f('not'[']) = 'not_(r) .

 eq '_:=_(k1,pk2) = pk2 -> f('_:='['])(k1) .
 eq r2 -> f('_:='['])(k1) = '_:=_(k1,r2) .

 eq 'if___(pk,k1,k2) = pk -> f('if'[']__)(k1,k2) .
 eq r -> f('if'[']__)(k1,k2) = 'if___(r,k1,k2) .

 eq 'halt_(pk) = pk -> f('halt'[']) .
 eq r -> f('halt'[']) = 'halt_(r) .
endfm
```

The `KResult` and `KProper` computations have precisely the same meaning as in the plain Maude approach. We use the `Qid` labels for K nodes corresponding to the original language syntax, and frozen labels for auxiliary freezers. Note that the module above did not need to import the language syntax, because it only refers to quoted identifier and frozen string labels.

Step 5 Define the configuration, if needed:

```
fmod IMP-K-CONFIGURATION is including STATE + K-META + IMP-SYNTAX + CONFIG .
  op k : K -> ConfigItem .
  op state : State -> ConfigItem .
  subsort Val < Config KResult .
  op [[_]] : Pgm -> Config .
  var p : Pgm .  var v : Val .  var c : KSet{ConfigItem} .
  eq [[p]] = [[k(mkK(p)) state(.State)]] .
  eq [[k(v) c]] = v .
endfm
```

The only difference between the module above and its corresponding one in the plain Maude approach is that the evaluation operator takes a program now instead of a computation, but lifts it with the provided operator `mkK` into a `KProper` computation when placing it into the configuration.

Step 6 Define the actual semantics. Like before, we should first "initialize the semantics" by establishing the connection between "terminals" in the syntax, i.e., meta-subterms with `Qid` labels starting with `'#`, and computations. One easy way to achieve that is to use `downK`, also provided as part of the prelude, as shown in the module below, which completes the semantics of our language using the K-meta approach:

```
mod IMP-K-SEMANTICS is including IMP-K-CONFIGURATION + IMP-K-STRICTNESS .
  vars k k1 k2 rest : K . var x : Name . var v : Val . vars i i1 i2 : Int . var b : Bool . var sigma : State .

  op int : Int -> Val .  op bool : Bool -> Val .  op name : Name -> KProper .
  eq '#int(k) = int(downTerm(k,errInt)) .     op errInt : -> [Int] .
  eq '#bool(k) = bool(downTerm(k,errBool)) .  op errBool : -> [Bool] .
  eq '#name(k) = name(downTerm(k,errName)) .  op errName : -> [Name] .

  rl k(name(x) -> rest) state(sigma) => k(sigma[x] -> rest) state(sigma) .
  rl '_+_(int(i1),int(i2)) => int(i1 + i2) .
  rl '_<=_(int(i1),int(i2)) => bool(i1 <= i2) .
  rl '_and_(bool(true),k2) => k2 .
  rl '_and_(bool(false),k2) => bool(false) .
  rl 'not_(bool(b)) => bool(not b) .
  rl k('_:=_(name(x),v) -> rest) state(sigma) => k(rest) state(sigma[x <- v]) .
  rl '_;_(k1,k2) => k1 -> k2 .
  rl 'skip.Stmt => .K .
  rl 'if___(bool(true),k1,k2) => k1 .
  rl 'if___(bool(false),k1,k2) => k2 .
  eq (k('while__(k1,k2) -> rest)).ConfigItem = k('if___(k1, (k2 -> 'while__(k1,k2)), .K) -> rest) .
  rl (k('halt_(int(i)) -> rest)).ConfigItem => k(int(i)) .
endm
```

Step 7 Test the semantics using several programs.

```
mod IMP-K-RUN is
  including IMP-PROGRAMS + IMP-K-SEMANTICS .
endm

rew [[pSum]] .
```

B.3 Comparing the Plain and Meta Approaches

Note that the one using the meta-level has several advantages: - it works directly with the syntax of the PL, so one does not need to first collapse everything into computations - it is slightly more compact, at least in number of lines and of operators defined. - it is more modular, in that one can add new features like self-generation of code, etc., without worrying of the actual language syntax.

The only advantage the core-level one has is that its rules look more readable.